



Persistent

Core Java: Multithreading

Persistent Interactive | Persistent University



Objectives :

At the end of this module, you will be able to :

- What is Multithreading ?
- Define Threads
- Differentiate between Process and Thread
- Explain types of Thread
- Understand the life cycle of a thread
- Explain java thread states and how to use it in Java
- Create threads
- Identify the thread priorities
- Understand thread synchronization and inter-threaded communication

Configurations required

- JDK 1.8
- Eclipse Luna
- MySQL
- Java 8 Documentation

Basics of Multithreading

- In this topics we will be going through what is multithreading, difference between thread and process, lifecycle of thread, java's thread model, ways to implement thread, methods of thread class ,what is synchronization? and finally inter-thread communications.

What is Multithreading...???

- Multithreading is executing the process into multiple threads simultaneously.
- Dividing a program into multiple parts and running them concurrently.
- Each part has assigned different responsibility and running them at the same time by making optimal use of resources.

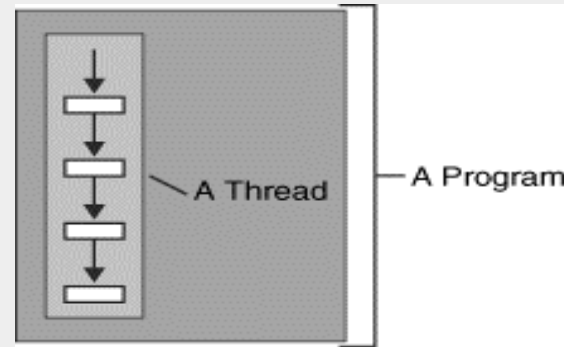
Multitasking - Running multiple processes on shared CPU.

Multithreading - Dividing a single process into parts i.e threads and running them parallel.

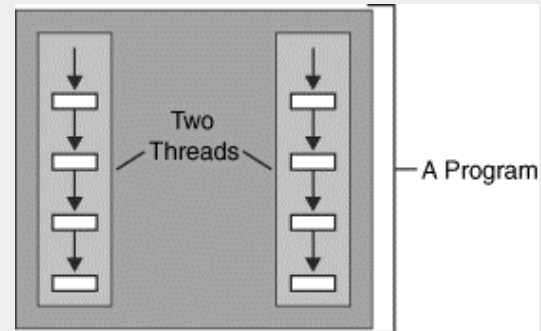
What is a Thread?

- A thread is a single sequential flow of control within a program.

- Single Threaded:



- Multithreading:



Overview

Process Based vs. Thread Based Multitasking:

| Multitasking:s | PROCESS BASED | THREAD BASED |
|--------------------|---------------|--------------|
| Smallest Unit: | Process | Thread |
| Overhead: | More | Less |
| Weight: | Heavy weight | Light weight |
| Context Switching: | Expensive | Inexpensive |
| Efficiency: | Less | More |

Why threads?

- Improved performance
- Minimized system resource usage
- Simultaneous access to multiple applications
- Program structure simplification
- Send & receive data on network
- Read & write files to disk
- Perform useful computation (editor, browser, game)

Java Thread Model

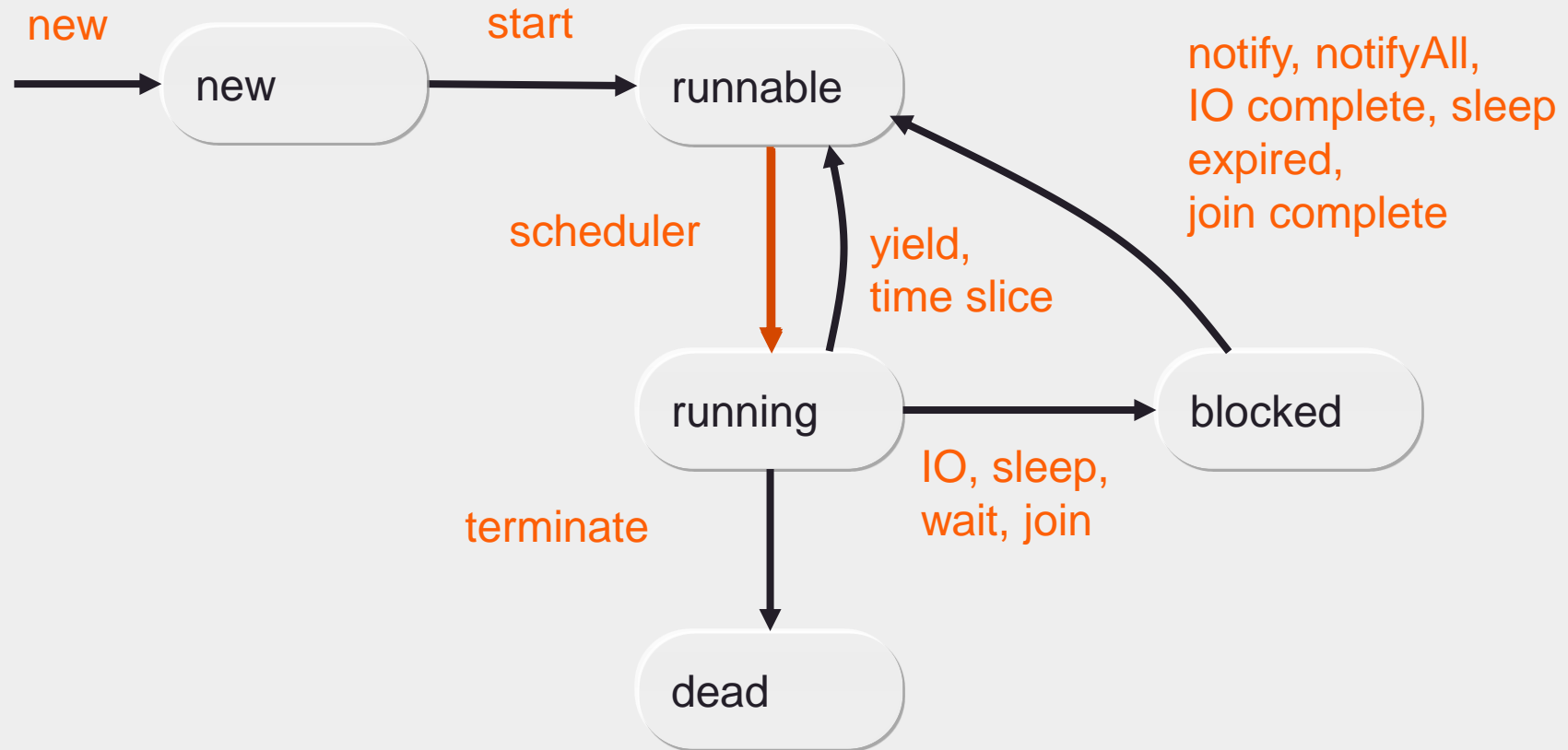
- Java provides support for threads even if the underlying operating system on which it runs does not.
- In the event that the operating system does not provide thread support, a package called "**green threads**" is ported together with the Java Virtual Machine to provide a threads layer on top of the operating system.
- Each thread has its own set of "virtual registers," also known as the thread's context; it's own stack; and a priority value.

Java Thread Model...

- Java provides built-in support for ***multithreaded programming***.
- Threads enable the entire environment to be asynchronous.
- preventing the waste of CPU cycles.
- Single-threaded systems use an approach called an ***event loop with polling***.
- In Java the main loop/polling mechanism is eliminated.
- One thread can pause without stopping other parts of the program.

Java thread states

State Diagram



Types of Threads

- Two types of threads:
 - USER THREADS
 - MAIN THREAD
 - Is automatically created to execute the main() method.
 - Should be the last thread to finish.
 - CHILD THREADS
 - Spawned from the main thread.

Types of Threads

- DAEMON THREADS

- The Java VM will not exit if non-Daemon threads are executing.
- “background” threads that provide services to other threads, e.g., the garbage collection thread.
- The Java VM will exit if only Daemon threads are executing.
- Daemon threads die when the Java VM exits.

The Main Thread

Output:Current thread:
Thread[main,5,main]

After name change:
Thread[My
Thread,5,main]

```
Thread t = Thread.currentThread();
```

```
System.out.println("Current thread: " + t);  
// change the name of the thread
```

```
t.setName("My Thread");
```

```
System.out.println("After name change: " + t);
```

```
try {
```

```
    for(int n = 5; n > 0; n--) {  
        System.out.print(n);  
        Thread.sleep(1000);
```

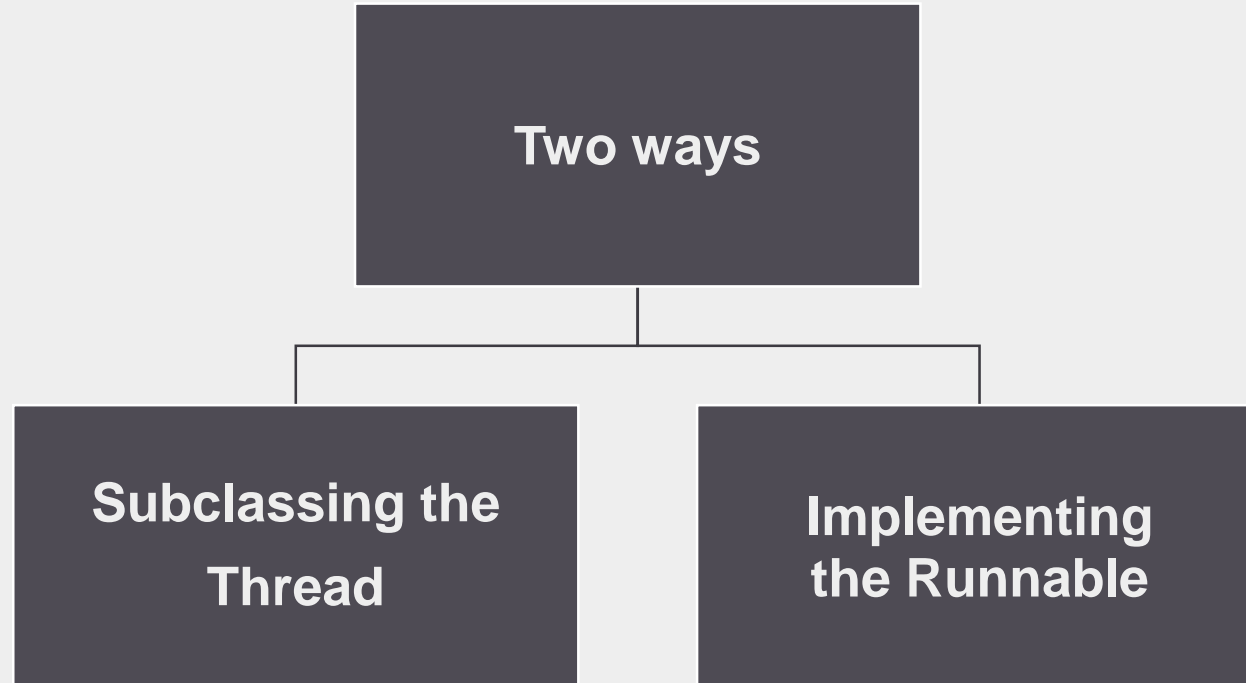
```
    }
```

```
catch (InterruptedException e) {
```

```
    System.out.println("Main thread interrupted");  
}
```

Creating Your Own Threads

- In both cases the **run()** method should be implemented



Thread Class & Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, Runnable.

| Method | Meaning |
|-------------|---|
| getName | Obtain a thread's name |
| getPriority | Obtain a thread's priority. |
| isAlive | Determine if a thread is still running. |
| join | Wait for a thread to terminate. |
| run | Entry point for the thread. |
| sleep | Suspend thread for a period of time. |
| start | Start a thread by calling its run method. |

Implementing the Runnable

- `run()` method defines an independent path of execution.
 - defines an ENTRY & EXIT
 - thread ends when `run()` method ends; either by normal termination or by throwing an uncaught exception.
- Procedure for creating threads:
 - Create a class that implements `Runnable`, implement only a single method `run()`.
 - Instantiate an object of type `Thread` from within that class. `Thread(Runnable threadOb, String threadName)`
 - Invoke `start()` method on the `Thread` object.

Creating threads (continued)

Implementing the Runnable Interface

```
class MyRunnable implements Runnable {  
    private String name, msg;  
  
    public MyRunnable(String name, String msg) {  
        this.name = name;  
        this.msg = msg;  
    }  
  
    public void run() {  
  
        System.out.println(name + " starts its execution");  
  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(name + " says: " + msg + " for “+i+”  
time/s”);  
  
            try {  
                Thread.sleep(2000);  
            }  
        }  
    }  
}
```

Creating threads (continued)

Implementing the Runnable Interface

```
catch (InterruptedException ie) {}  
    } // End of For Loop  
    System.out.println(name + " finished execution");  
    }  
}
```

```
public class TestMyRunnable  
{  
    public static void main(String[] args)  
    {  
        MyRunnable myrun = new MyRunnable("PSL",  
        "Hello");  
        Thread mythread = new Thread(myrun);  
        mythread.start();  
    }  
}
```

Creating threads (continued)

Implementing the Runnable Interface

A New
Runnable
object is
created



```
public class TestMyRunnable
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        MyRunnable myrun = new MyRunnable("PSL", "Hello");
```

```
        Thread mythread = new Thread(myrun);
```

```
        mythread.start();
```

```
    }
```

```
}
```

Creating threads (continued)

Implementing the Runnable Interface

A New
Thread is
created



```
public class TestMyRunnable
{
    public static void main(String[] args)
    {
        MyRunnable myrun = new MyRunnable("PSL", "Hello");
        Thread mythread = new Thread(myrun);
        mythread.start();
    }
}
```

Creating threads (continued)

Implementing the Runnable Interface

```
public class TestMyRunnable
{
    public static void main(String[] args)
    {
        MyRunnable myrun = new MyRunnable("PSL",
        "Hello");

        Thread mythread = new Thread(myrun);

        mythread.start();

    }
}
```

A newly created thread is moved either to runnable or running state by the scheduler

Creating threads (continued)

- **Output**

PSL strats its execution

PSL says : Hello for 1 time/s

PSL says : Hello for 2 time/s

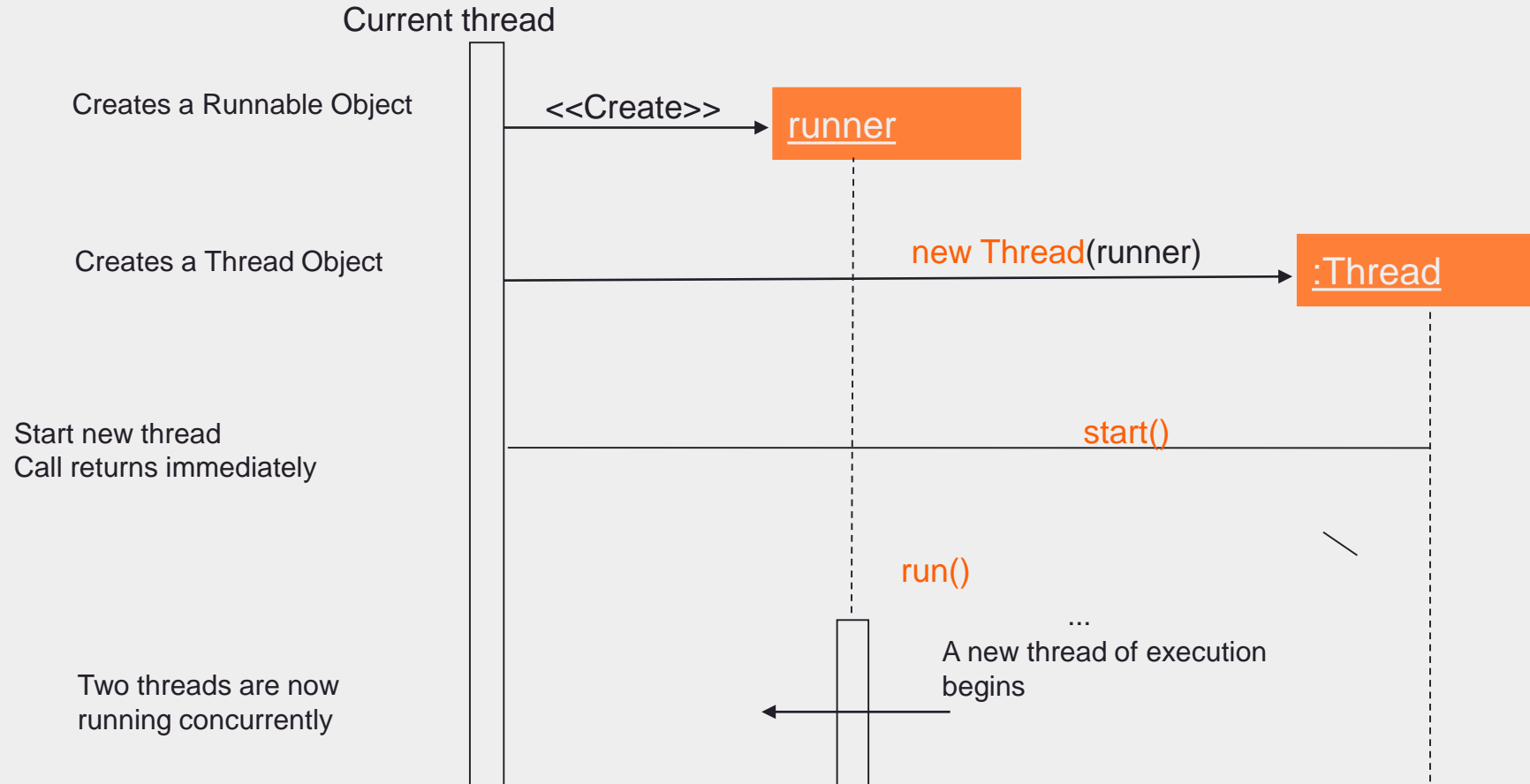
PSL says : Hello for 3 time/s

PSL says : Hello for 4 time/s

PSL says : Hello for 5 time/s

PSL finished execution

Implementing Runnable : Steps

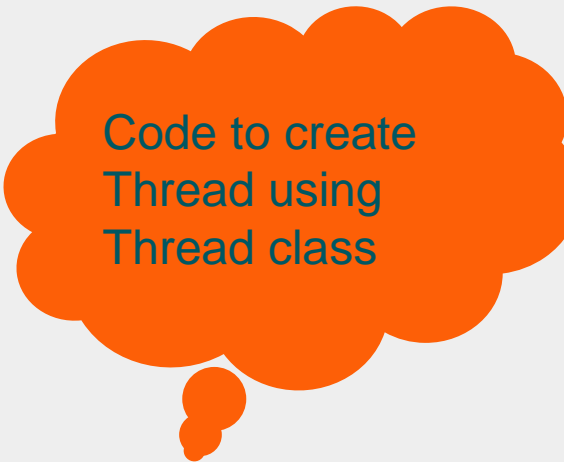


Extending the Thread Class

- Procedure for creating threads:
 - Create a class that extends Thread class & overrides the run() method.
 - Call a Thread constructor explicitly in its constructors to initialize the thread (using the super()).
 - Invoke the start() method on the object of the class.

Creating threads (continued)

Extending from java.lang.Thread class



Code to create
Thread using
Thread class

```
class MyThread extends Thread {  
    private String name, msg;  
  
    public MyThread(String name, String msg) {  
        this.name = name;  
        this.msg = msg;  
    }  
  
    public void run() {  
        System.out.println(name + " starts its execution");  
        for (int i = 0; i <= 5; i++) {  
            System.out.println(name + " says: " + msg + " for “+i+”  
time/s”);  
            try {  
                Thread.sleep(2000);  
            }  
            catch (InterruptedException ie) {}  
        } // End of For Loop  
        System.out.println(name + " finished execution");  
    }  
}
```

Creating threads (continued)

Extending from java.lang.Thread class

```
class MyThread extends Thread {  
    private String name, msg;  
    public MyThread(String name, String msg) {  
        this.name = name;  
        this.msg = msg;  
    }  
  
    public void run() {  
  
        System.out.println(name + " starts its execution");  
        for (int i = 0; i <=5; i++) {  
            System.out.println(name + " says: " + msg+” for “+i+”  
time/s”);  
            try {  
                Thread.sleep(2000);  
            }  
        }  
        catch (InterruptedException ie) {}  
        }// End of For Loop  
        System.out.println(name + " finished execution");  
    }  
}
```

Creating threads (continued)

Extending from java.lang.Thread class

Overridden Method

```
class MyThread extends Thread {
    private String name, msg;

    public MyThread(String name, String msg) {
        this.name = name;
        this.msg = msg;
    }

    public void run() {
        System.out.println(name + " starts its execution");
        for (int i = 1; i <= 5; i++) {
            System.out.println(name + " says: " + msg + " for " + i + "
time/s");
            try {
                Thread.sleep(2000);
            }
            catch (InterruptedException ie) {}
        } // End of For Loop
        System.out.println(name + " finished execution");
    }
}
```

Creating threads (continued)

Extending from java.lang.Thread class

It will throw an
InterruptedException



```
class MyThread extends Thread {  
    private String name, msg;  
    public MyThread(String name, String msg) {  
        this.name = name;  
        this.msg = msg;  
    }  
    public void run() {  
        System.out.println(name + " starts its execution");  
        for (int i = 1; i <=5; i++) {  
            System.out.println(name + " says: " + msg+" for  
            "+i+" time/s");  
            try {  
                Thread.sleep(2000);  
            }  
            catch (InterruptedException ie) {}  
            }// End of For Loop  
        System.out.println(name + " finished execution");  
    }  
}
```

Creating threads (continued)

Extending from `java.lang.Thread` class

A New Thread
Created But
Not Running



```
public class TestMyThread
{
    public static void main(String[] args)
    {
        MyThread mth1 = new MyThread("PSL", "Hello");
        mth1.start();
    }
}
```

Creating threads (continued)

Extending from java.lang.Thread class

run() method is called. The New Thread moves to runnable state or to the running state depending upon the scheduler.



```
public class TestMyThread
{
    public static void main(String[] args)
    {
        MyThread mth1 = new MyThread("PSL", "Hello");
        mth1.start();
    }
}
```

Creating threads (continued)

Extending from `java.lang.Thread` class

Can U Predict The Output of the Program

Creating threads (continued)

- **Output**

PSL starts its execution

PSL says : Hello for 1 time/s

PSL says : Hello for 2 time/s

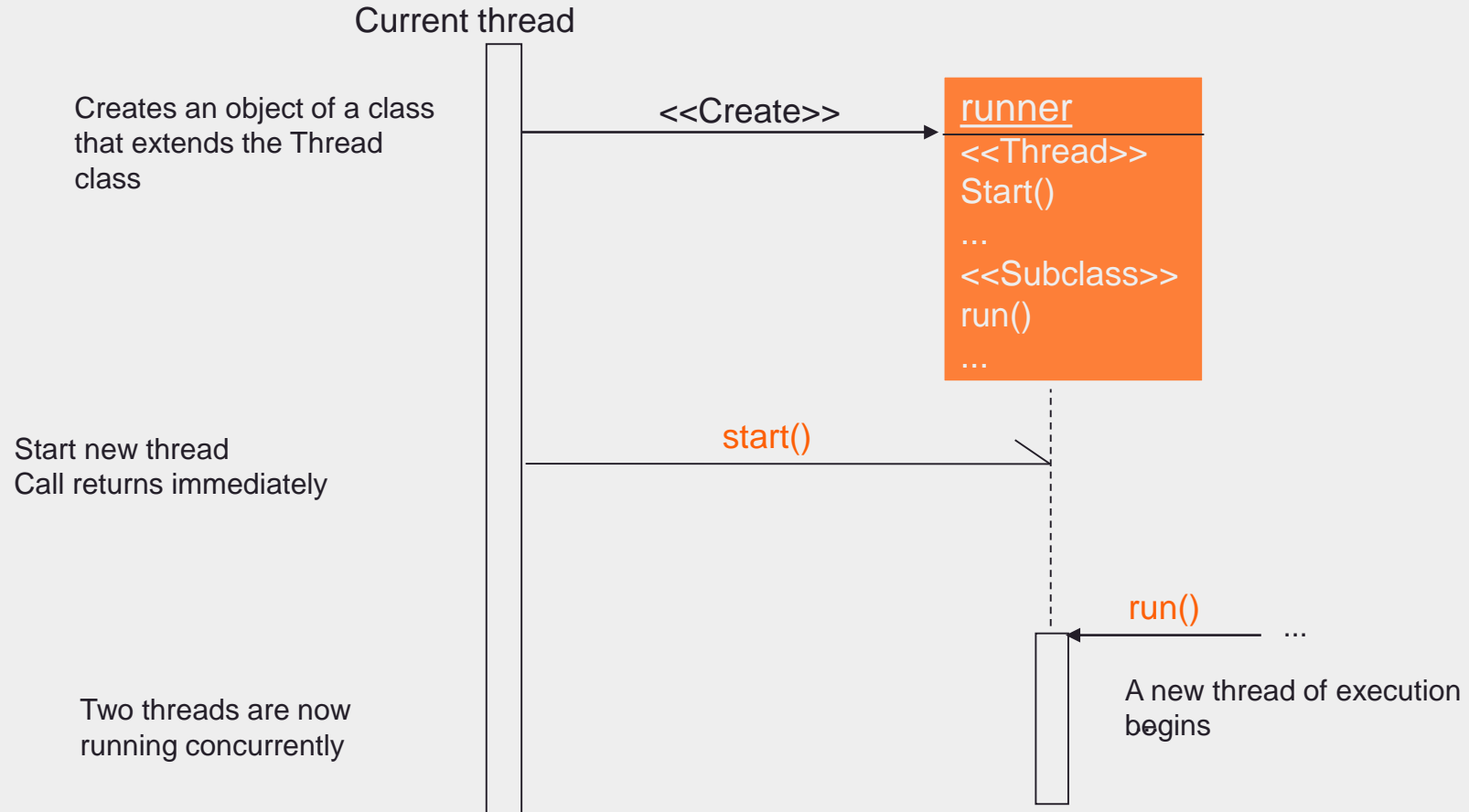
PSL says : Hello for 3 time/s

PSL says : Hello for 4 time/s

PSL says : Hello for 5 time/s

PSL finished execution

Extending the Thread Class: Steps



Which to Choose?

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
- Logically you may want to extend Thread only if you are customizing the Thread characteristics.

Thread Methods

- `final boolean isAlive()`
- `final int getPriority()`
`final void setPriority(int newPriority)`
- `static void yield()`
- `static void sleep(long millisec)`
- `final void join()` throws `InterruptedException`
`final void join(long millisec)` throws `InterruptedException`
- `void interrupt()`

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- higher-priority threads get more CPU time than lower-priority threads.
- A higher-priority thread can also preempt a lower-priority one.
- When a thread is created, it inherits its priority from the thread that created it.

MIN_PRIORITY : 1

MAX_PRIORITY : 10

NORM_PRIORITY : 5

Scheduling

- Threads run concurrently ... really?
- Thread **scheduling** is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either **preemptive** or **nonpreemptive**.
 - **Preemptive scheduling** – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
 - **Nonpreemptive scheduling** – the scheduler never interrupts a running thread (relies on the running thread to yield control of the CPU)

Java Scheduling

- Scheduler is **preemptive** and **based on priority** of threads.
- Uses **fixed-priority scheduling**:
 - Threads are scheduled according to their priority w.r.t. other threads in the ready queue
 - The highest priority runnable thread is always selected for execution above lower priority threads
 - When multiple threads have equally high priorities, only one of those threads is guaranteed to be executing
 - Java threads are guaranteed to be preemptive-but not time sliced

Some Trivia

- Some Windows operating systems support only 7 priority levels, so different levels in Java may actually be mapped to the same operating system level.

Moral: Don't base your system logic on thread priority

Synchronization

- In this topics we will see how could we handle multiple threads when they try to access same resource at the same times and that is possible by using synchronization or concurrency control.

Concurrency Control and Synchronization

- A **shared resource** in a program can be changed by more than one thread.
- Java provides a language support to ensure that only one thread accesses the resource – Synchronization
- Achieved in java by:
 - Adding the **synchronized keyword** to a method
 - Using a **synchronized block**.
- Key to **synchronization** is the concept of **monitor**.

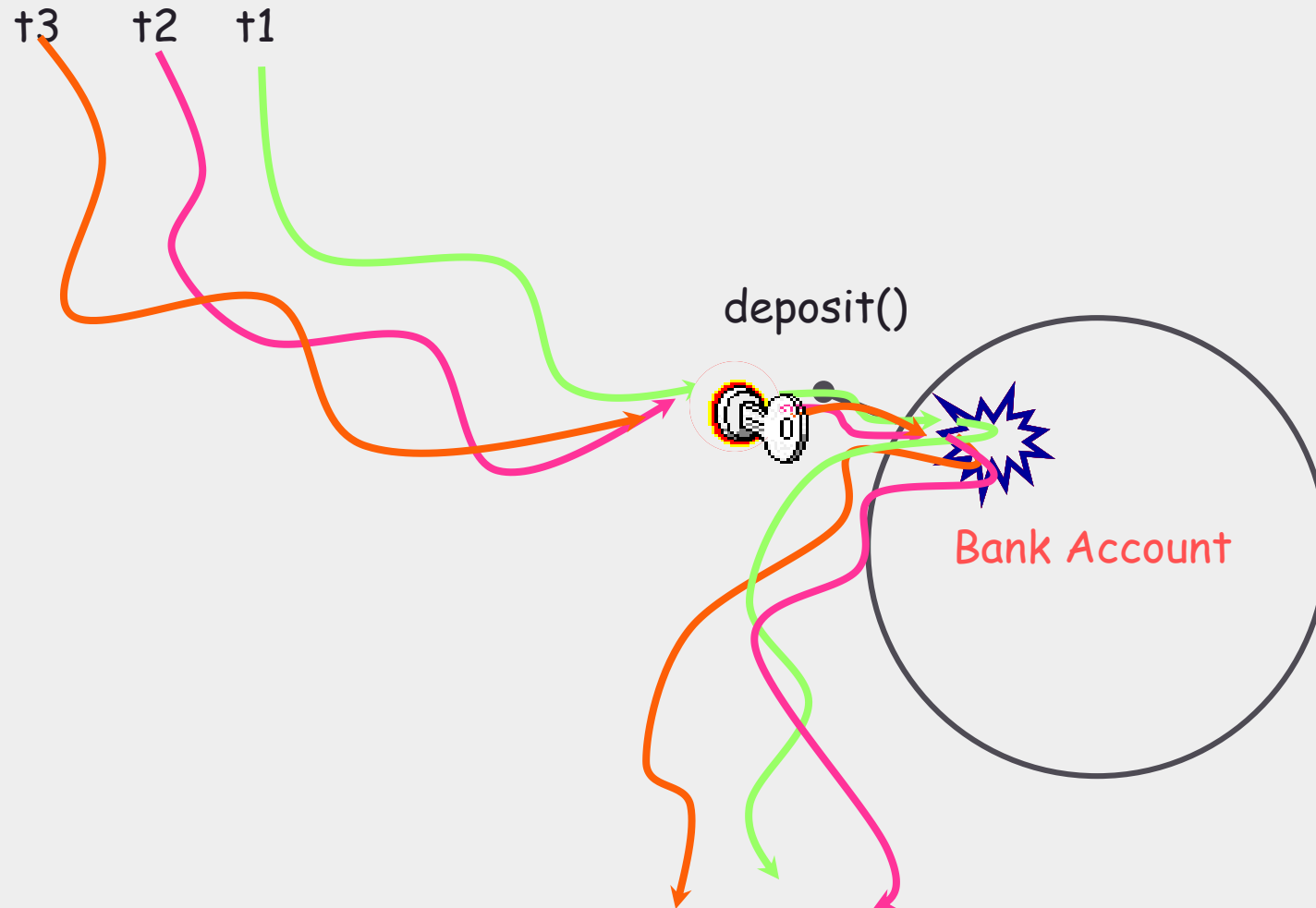
Monitor or Semaphore

- Each object has a **monitor** that is a token used to determine which application thread has control of a particular object instance.
- Only one thread can **own** a monitor at a given point of time.
- When a thread acquired a lock it is said to have **entered** the monitor.
- All other threads, are suspended, till the first thread **exits** the monitor.
- Access to the object monitor is **queued**.

Synchronized Method ...E.g.

```
public class BankAccount {  
  
    private float balance;  
  
    public synchronized void deposit(float amount) {  
        balance += amount;  
    }  
  
    public synchronized void withdraw(float amount) {  
        balance -= amount;  
    }  
  
}
```

Example (cont.)

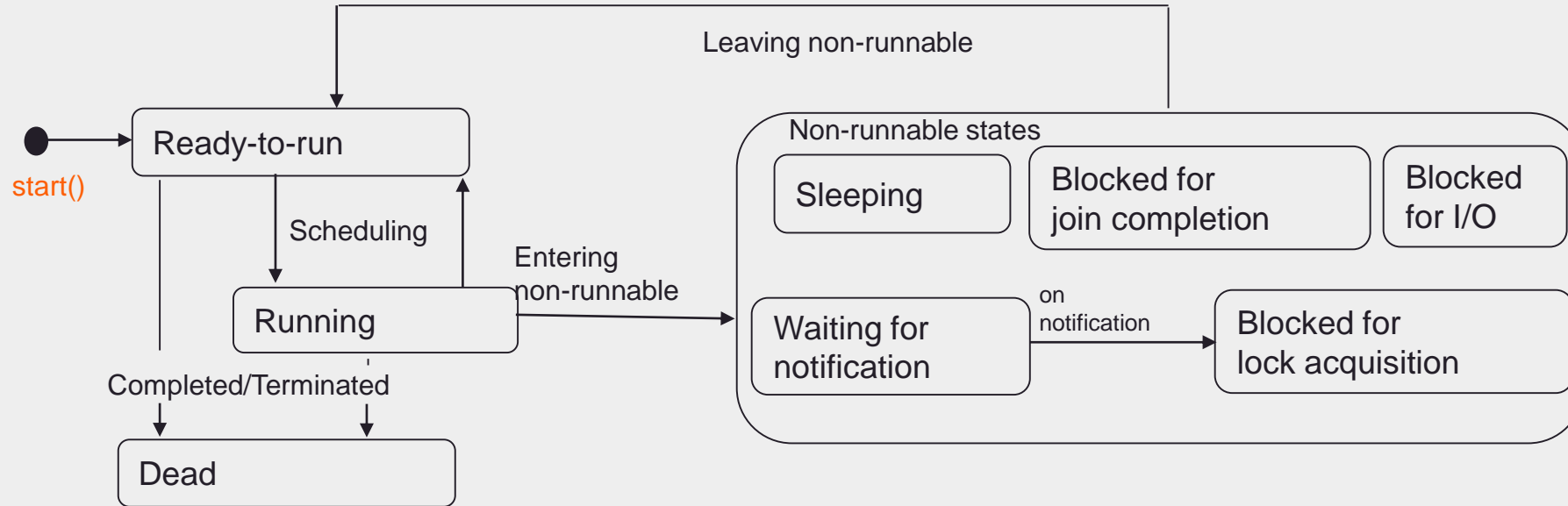


Synchronized Blocks

- Can be used when method cannot be marked synchronized (e.g. 3rd party classes).
- Performance benefit by marking only critical sections as synchronized.

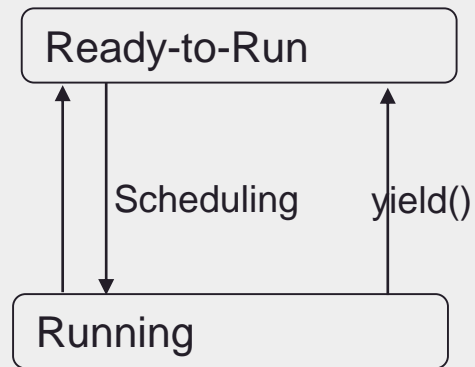
```
class SmartClient{  
    BankAccount account;  
    // ...  
    public void updateTransaction(){  
        synchronized (account){//synchronized block  
            account.update();  
        }  
    }  
}
```

Thread Transitions



Running & Yielding

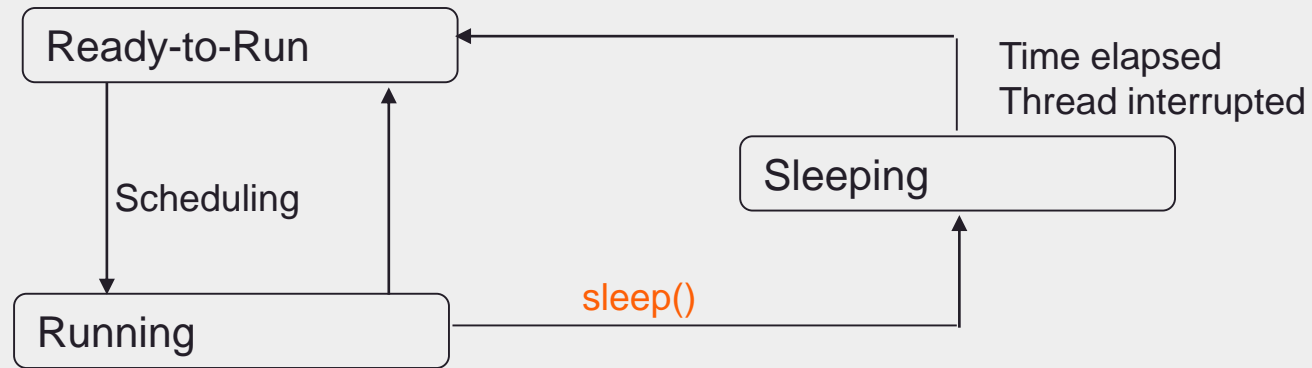
- After start() method has been called, the thread starts life in the Ready-to-run state.
- Once in the Ready-to-run state, the thread is eligible for running i.e. it waits for its turn to get CPU time.
- The thread scheduler decides which thread gets to run and for how long.



```
public void run(){  
    try {  
        while(!done()){  
            doLittleBitMore();  
            Thread.yield(); //thread yields  
        }catch(InterruptedException e){  
            doCleaningUp();  
        }  
    }  
}
```


Sleeping & Waking up

- `sleep()`
 - causes the currently running thread to pause its execution & transit to Sleeping state.
 - does not relinquish any lock that the thread might have.
 - thread will sleep for at least the time specified in the argument.
 - if a thread is interrupted while sleeping, it will throw an ***InterruptedException*** when it awakes & gets to execute.



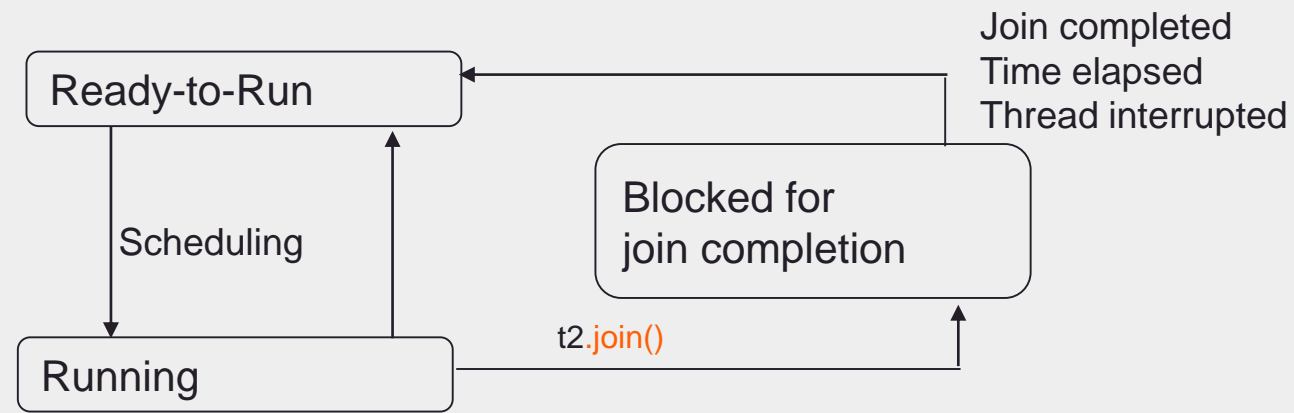
Blocking for I/O

- A blocking thread, on executing a *blocking operation* requiring a resource (like a call to an I/O method), will transit to the Blocked-for-I/O state.
- A blocking operation must complete before the thread can proceed to the Ready-to-run state.
- E.g. - a thread reading from standard input terminal, which blocks until input is provided.

```
int input = System.in.read();
```

Joining

- **join()**
 - A thread invoke join() on another thread in order to wait for the other thread to complete its execution before continuing.
 - means first thread waits for the second thread to join it after completion.



Inter-thread communication

- In this topics we will see how threads can communicate with each other to achieve some task efficiently and faster i.e. how to work with `wait()`, `notify()`, `notifyAll()` methods from `object` class.

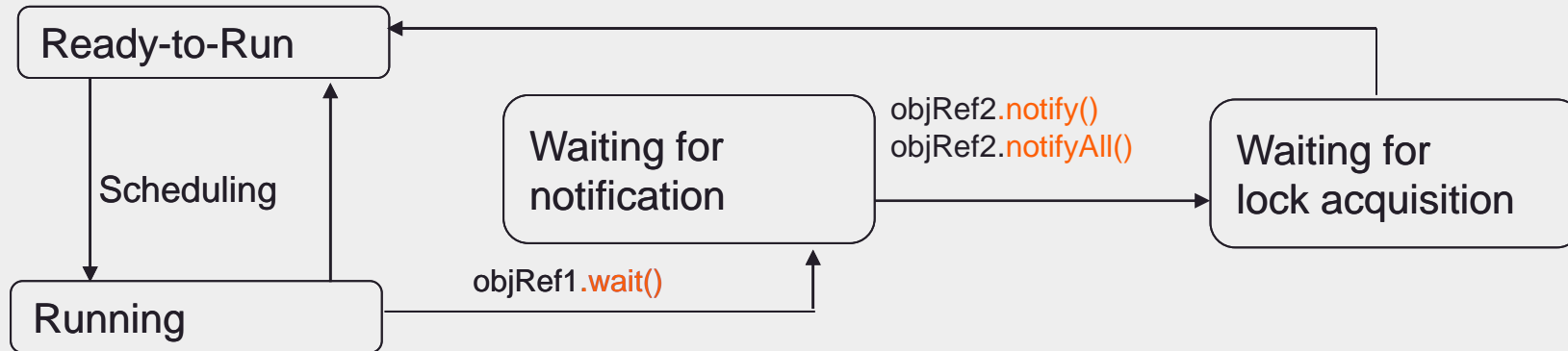
Interthread communication

A thread may notify another thread that the task has been completed. This communication between threads is known as interthread communication.

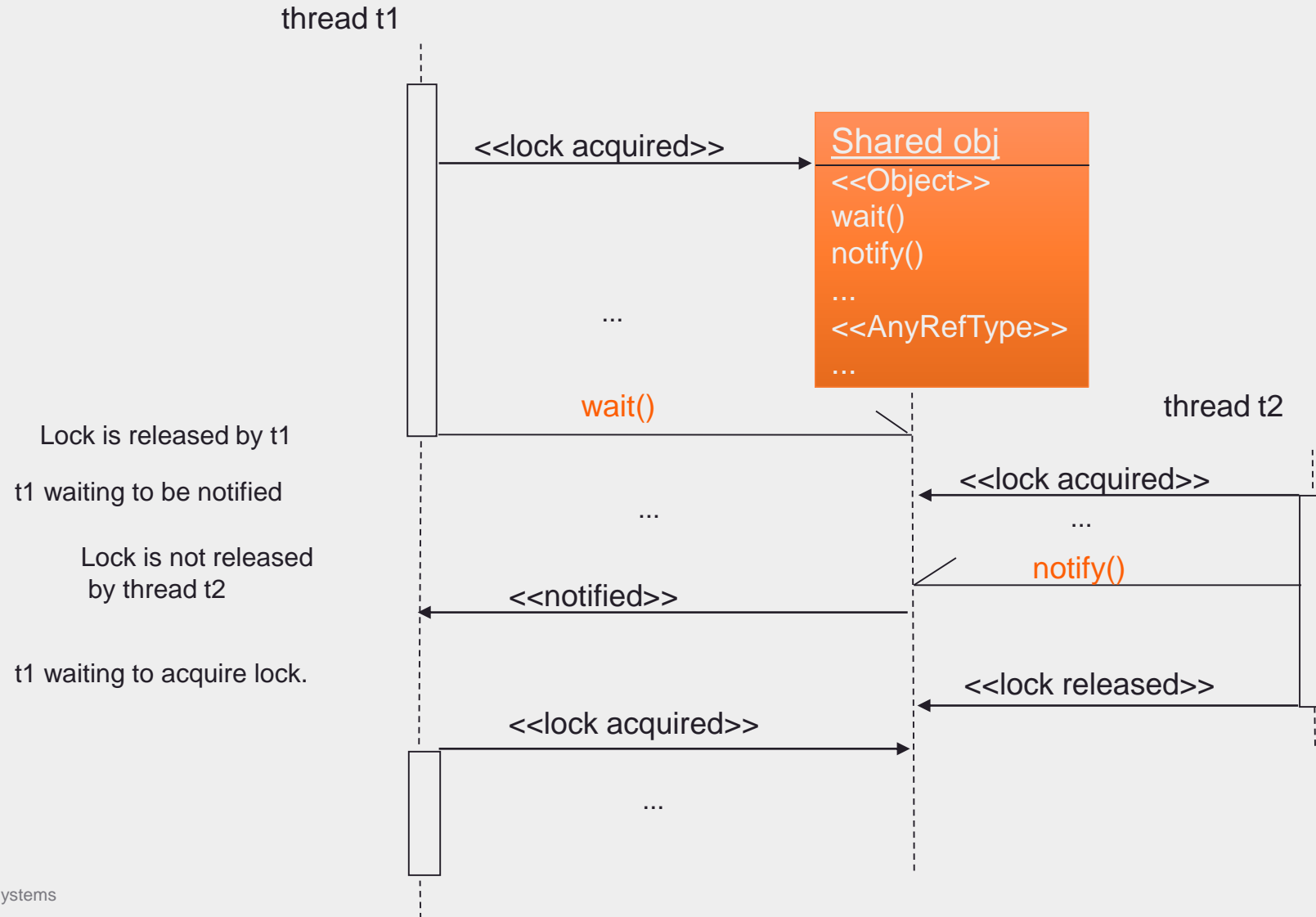
- The various methods used in interthread communication are:
 - ✓ wait()
 - ✓ notify()
 - ✓ notifyAll()

Interthread Communication: Waiting & Notifying

- **wait()**, **notify()**, and **notifyAll()** methods.
 - all three methods can be called only from within a **synchronized** method.



Wait and notify: Steps



Thread Termination

- A thread can transit to the Dead state from the ***Running*** or the ***Ready-to-run*** states.
- The thread dies when it completes its run method, either by returning normal or by throwing an exception.
- Once in this state, the thread can not be resurrected.
- There is no way the thread can be enabled for running again, not even by calling the start() method once more on the thread object.

Thread Groups

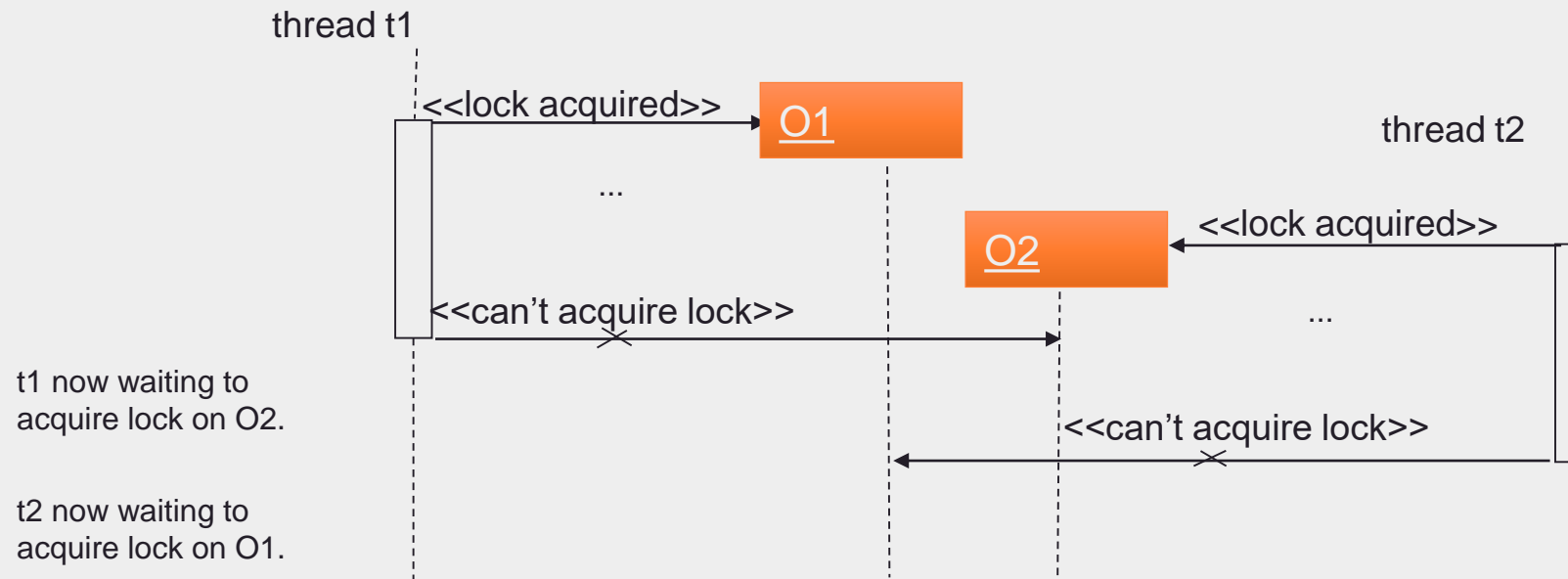
- Mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- These are implemented by the **ThreadGroup** class.
- You cannot move a thread to a new group after the thread has been created.
- **The Default Thread Group : *main()***

Thread Group example

```
public class ThreadGroupDemo implements Runnable {  
    public void run() {  
  
        System.out.println(Thread.currentThread().getName()  
        ));  
    }  
  
    public static void main(String[] args) {  
        ThreadGroupDemo runnable = new  
        ThreadGroupDemo();  
        ThreadGroup tg1 = new ThreadGroup("Parent  
        ThreadGroup");  
        Thread t1 = new Thread(tg1, runnable, "one");  
        t1.start();  
        Thread t2 = new Thread(tg1, runnable, "two");  
        t2.start();  
        Thread t3 = new Thread(tg1, runnable, "three");  
        t3.start();  
        System.out.println("Thread Group Name: " +  
        tg1.getName());  
    }  
}
```

Deadlocks

- A situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.
- Both remain waiting forever in the ***Blocked-for-lock-acquisition*** state (called as ***deadlock***).



FAQ

- What is Multithreading and Multitasking?
- Describe Thread Lifecycle ?
- How to Create a Threads?
- What is runnable state?
- What is Synchronization?
- How to achieve inter-thread communication?
- How to overcome Deadlock situation?

Summary

With this we have come to an end of our session, where we discussed about

- Multithreading
- Concurrency control mainly

At the end of this session, we see that you are now able to answer following questions:

- What is Multithreading?
- What is thread lifecycle?
- What is thread concurrency and inter-thread communication?

Appendix

A decorative graphic consisting of a horizontal orange line that extends from the left edge of the slide to the right. At the right end of this line, a vertical orange line descends downwards. A large orange circle is positioned such that its bottom edge is tangent to the horizontal line, and its left edge is tangent to the vertical line.

References

Thank you

Reference Material : Websites & Blogs

- <http://www.javatpoint.com/multithreading-in-java>
- <http://javahungry.blogspot.com/2013/07/threads-lifecycle-example-java-methods-explanation.html>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/>
- <http://tutorials.jenkov.com/java-concurrency/index.html>
- http://www.tutorialspoint.com/java/java_thread_communication.htm

Reference Material : Books

- **Head First Java**
 - By: Kathy Sierra, Bert Bates
 - Publisher: O'Reilly Media, Inc.
- **Java Complete Reference**
 - By Herbert Schildt



Thank you!

Persistent Interactive | Persistent University

